Component-based modeling and observer-based verification for railway safety-critical applications

Marc Sango, Laurence Duchien, Christophe Gransart

Component-Based Modeling and Observer-Based Verification for Railway Safety-Critical Applications

Marc Sango[1], Laurence Duchien[2], and Christophe Gransart[1]

[1] Univ Lille Nord de France, F-59000 Lille, IFSTTAR, LEOST, F-59650, France
{marc.sango, christophe.gransart}@ifsttar.fr
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, University of Lille 1, France
{laurence.duchien}@inria.fr

Abstract. One of the challenges that engineers face, during the development process of safety-critical systems, is the verification of safety application models before implementation. Formalization is important in order to verify that the design meets the specified safety requirements. In this paper, we formally describe the set of transformation rules, which are defined for the automatic transformation of safety application source models to timed automata target models. The source models are based on our domain-specific component model, named SARA, dedicated to SAfety-critical RAilway control applications. The target models are then used for the observer-based verification of safety requirements. This method provides an intuitive way of expressing system properties without requiring a significant knowledge of higher order logic and theorem proving, as required in most of existing approaches. An experimentation over a chosen benchmark at rail-road crossing protection application is shown to highlight the proposed approach.

Keywords: software component, timed automata, transformation, verification, safety-critical applications

1 Introduction

Safety-critical systems must conform to safety standards defined by domain standardizations, such as the European standard of software for railway control and protection systems, EN 50128 [8]. This is why one of the challenges that engineers face, during the development process of safety-critical systems, is the verification of safety application models before implementation. Over the last few years, the complexity of safety applications has increased. Then the modeling and the formalization of safety applications is becoming a very difficult task.

Component-Based Software Engineering (CBSE) is a possible software modeling solution. It is an established approach for modeling a complex software and for facilitating integration by a third party [20]. There are several research works that facilitate component-based development in general or in domain-specific purposes [5, 2, 10, 21]. In this paper, we use our domain-specific component model, named SARA, dedicated to the development of SAfety-critical

RAilway control applications [17]. The objective of our approach consists in modeling and enforcing dependability requirements during the development process of safety-critical applications in order to facilitate their certification process.

On the other hand, design models have to be mapped to formal models for automatic verification. In this work, we focus on verification approaches that take advantage of the flexibility of reliable component models and analysis facilities offered by formal models in order to satisfy timing requirements. There are several research works that propose the transformation of informal or semi-formal models into formal models, which are supported by available verification tools [1, 4, 15]. For example, for the safety of rail-road protection systems, Mekki et al. use the model-driven architecture approach to systematically transform the UML state machine into the Timed Automata (TA) in order to validate some temporal requirements [15]. In this approach, the three-tier approach for the composition of pre-verified components is not explicitly considered. Based on pre-defined formal models of source and target models, Soliman et al. transform the function block diagram to the Uppaal timed automata [19]. In the same way, Bhatti et al. suggest an approach for the verification of IEC 61499 function block applications by using the observer-based verification [4].

In this paper, we focus on the transformation phase from our SARA model to the TA model, which is one of the most popular models adapted for the verification of timing properties [3]. The transformation algorithm consists of transformation rules. Both the source and the target domain models have been formally defined. Then, these formal definitions are used for the definition of transformation rules. The target models and the timing requirement observers are next used for the observer-based verification of safety applications. This method provides an intuitive way of expressing system properties without requiring a significant knowledge of higher order logic and theorem proving, as required in most of the existing approaches [4]. Indeed, users can use predefined observer patterns or can enhance them for their verification tasks.

The remainder of this paper is organized as follows. In Section 2, we motivate our approach by an example of rail-road crossing protection system. In Section 3, we give an overview of the suggested process of modeling and verification. In section 4, we introduce the formal definitions of the SARA component model and the formal definitions in terms of the TA formalism. Then, based on these definitions, the transformation rules are defined. In Section 5, based on a use case of our motivating example, the observer patterns of some safety requirements are presented. These observer models are synchronized with the use case model to do the verification. Finally, we discuss related works in Section 6 before concluding and pointing out future directions of our research in Section 7.

2 Motivating example

Our approach is motivated by using a rail-road intersection protection system. Fig. 1 presents an implementation of a rail-road level crossing control application by using software components. Note that, in this section, we focus on its global

description. The elements, component types and component connections types will be detailed in Section 4.1. The component *Sensor* embedded in the train reads information from track sensors to detect an approaching and an exiting of trains in the monitored intersection. This information is translated into a sequence of events *appr* and *exit* depending on the distance *dist* of trains from the intersection. The intersection gate is operated through the *Gate* component which closes or opens the gate. The *Gate* component responds to events by moving barriers *up* or *down*. The component *Controller* acts as a mediator between the other two components. It receives events from the sensor component and decides to open or not the gate to road traffic.

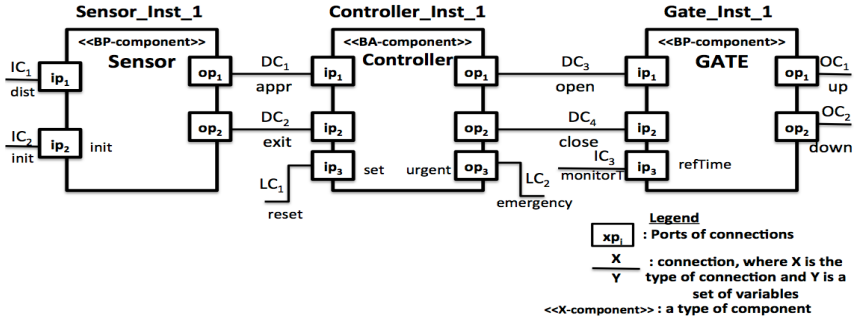

Fig. 1. The example of SARA components

For example, when the approach signal *appr* is received (respectively *exit*), the component instance $Controller\_Inst\_1$ sends immediately a *close* signal (respectively *open*) to the gate instance $Gate\_Inst\_1$. In general, the behavior of controller instance depends on operation rules, i.e., in some operation scenarios. In fact, railway level crossing behavior depends generally on the national operation rules [22]. But in most European countries, the automatic protection system gives absolute or relative priority to railway traffic, while preventing road users from crossing whenever a train is approaching [15]. In this work, we use an adapted use case, which is presented in detail in Section 5.

It is possible that the interaction of components in Fig. 1 results in the violation of system safety requirements specification, such as:

Requirement 1: "the gate must be down whenever a train is inside the rail-road crossing section" (adapted from [4]);

Requirement 2: "when the gate is opened to road traffic, it must stay open at least T_{min} time units, where T_{min} represents the minimum desired period of time separating two successive closing cycles of gate" (adapted from [15]);

Requirement 3: "once closed and when there is no train approaching meanwhile, the gate must be kept closed at least (T_{begin}) and at most (T_{end}), where T_{begin} and T_{end} are the time limits prescribed" (adapted from [15]).

The above application architecture and these safety requirements under verification are used throughout the rest of the paper.

3 Overview

Fig. 2 shows the schematic structure of our component-based modeling and observer-based verification approach. It is composed of two development paths and one verification tool.

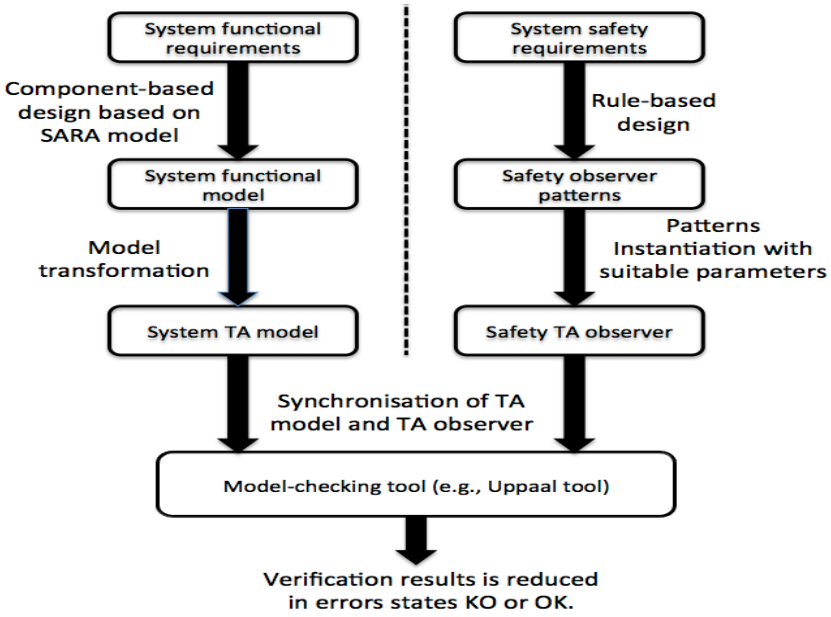

Fig. 2. Our methodology

The path on the left of Fig. 2 represents the system functional development path. It is responsible for performing the functional requirements. Functional requirements are modeled according to a component-based paradigm. In this paper, we use our SARA component model. Then, this model is translated into the TA formal model. One of the main parts of our method is the transformation of the SARA model to the TA model, as detailed in Section 4. The result of this path is a system TA model.

The path on the right of Fig. 2 represents the system safety development path. It is responsible for monitoring safety requirements. Generally, safety requirements depend on operation rules developed with a rule-based paradigm. For each safety requirement, the appropriate observation pattern is selected from the generic patterns and then instantiated to produce a corresponding safety requirement observer, as detailed in Section 5. The result of this development path is a safety TA observer.

In the end, the safety TA observers instantiated are synchronized with the system TA model obtained to generate a system global TA model. Then, by using a verification tool (e.g., Uppaal model checkers), the verification task is reduced to a reachability search of an error or no-error states (KO or OK states) on the global TA model.

4 From SARA model to TA model

In order to define the transformation rules, we first formally define our source model, i.e., the SARA model and the target model, i.e., the TA model. Then, based on these two formal models, we define the transformation rules from the SARA model to the TA model for the verification of safety-critical applications.

4.1 SARA model

SARA component model is a domain-specific component model dedicated to SAfety-critical RAilway control applications [17]. Its Application Programming Interface (API), which is specified with Ravenscar profile of Ada programming language, is defined to implement the train speed supervision application [16].

According to the SARA model and its Ravenscar API, a component specification is defined as an entity which encapsulates data structures with operations working on these data. The component specification (a) is separated from the component body (b), from the component instance (c) and from the component runtime (d) (see Fig. 3). Firstly, each component specification is distinguished by a unique name (e.g., CompTypeName in line 1 of Fig. 3.(a)). Each component specification defines the interface of operations for its instances by a set of input parameters, a set of output parameters or input/output parameters (e.g., see line 10 in Fig. 3.(a)). Operations are annotated with the timing requirement annotations (e.g., see lines 12-15 in Fig. 3.(a)). These annotations can be checked, as illustrated in Fig. 3.(d) with the Ada language annotations checking tool [9]. However, tasks and synchronization are not currently permitted in this tool.

Secondly, in the body of components, the behavior of component instances is defined by users (e.g., see lines 5-6 Fig. 3.(b)). Thirdly, SARA application, i.e., a top-level component named SARAProg, built from others connected components, can only be instantiated when the required resources are present (e.g., see lines 17-18 in Fig. 3.(c)), while components can only be instantiated within an application or other components (e.g.; see lines 7-15 in Fig. 3.(c)). Finally, Fig. 3.(d) shows the screen shot of the runtime execution and runtime annotation checking of our application implementation based on SARA component model.

More formally, the SARA component model is defined as follows.

Definition 1 (SARAProg). A SARA program is defined as a tuple SARAProg = (ProgName, IP, OP, LV, CV, IO, ProgBody), where:

- ProgName is the name of the program, which is defined by its developer;
- IP is a set of input parameters, which enter the input ports of program components, e.g., IP = $\{dist, init, reset, monitorT\}$ in Fig. 1;
- OP is a set of output parameters, which exit the output ports of program components, e.g., OP = $\{up, down, emergency\}$ in Fig. 1;
- LV is a set of local variables of this program. For technical reasons, we assume that all the local variables of program components occur somewhere in the program structure. e.g., LV = $\{init, set, urgent, refTime\}$ in Fig. 1;

(a)

```
1  package CompTypeName is
2    -- Component type:
3    type CompType is new Sara.Comps↩
         with record
4      Time_Var : Integer;
5      Train_Pos: Float;
6    end record;
7    -- other data:
8    Tmax : constant Integer := 15;
9    -- operations:
10   Procedure Op (C : in out ↩
         CompType)
11   -- operation annotations:
12   with
13   pre => (C.Time_Var = 0),
14   Post => (C.Time_Var <= Tmax
15     and then C.Train_Pos=10.0);
16 end CompTypeName;
```

(b)

```
1  package body ↩
       CompTypeName is
2    Procedure Op (C : in↩
         out CompType) ↩
       is
3    begin
4    -- user defined ↩
         behavior: e.g.,
5    C.Time_Var := 10;
6    C.Train_Pos := 12.0;
7    end Op;
8  end CompTypeName;
```

(c)

```
1  with CompTypeName;
2  Procedure SARAProg -- Top-level component
3    -- prog input parameters:
4    Train_Init : Integer := 0;
5    Train_X_Coord : Float := 0.0;
6    -- prog component instances:
7    CompInst : Sara.Prog.CompType :=
8    (Time_Var => Train_Init,
9     Train_Pos => Train_X_Coord);
10   -- program component execution:
11   package Defined_Task is new Sara.↩
         Periodic_Task
12           (Comp_Type => Sara.Prog.CompType,
13            Comp_Inst => Sara.Prog.CompInst,
14            Priority => 1, Period => 0.010,
15            Provided_Op => CompTypeName.Op);
16   -- program resources:
17   use Defined_Task;
18   use Ada.Calendar; -- or Ada.Real_Time
19   Start, Finish : Time;
20 begin
21   Start := Clock;  -- get system start time
22   ...
23 end SARAProg;
```

(d)

```
test — bash — 103×13
macbook-pro-de-christophe-gransart:test marcsango$ gnatprove -P SARAProg.gpr --report=all
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: translation to intermediate language ...
Statistics logged in /Users/marcsango/Documents/papers/FACS-2014/test/gnatprove/gnatprove.out
(detailed info can be found in /Users/marcsango/Documents/papers/FACS-2014/test/gnatprove/*.alfa)
Phase 3 of 3: generation and proof of VCs ...
analyzing SARAProg, 1 checks
saraprog.adb:13:16: info: precondition proved
analyzing CompTypeName.Op, 1 checks
comptypename.ads:15:15: postcondition not proved
analyzing precondition for CompTypeName.Op, 0 checks
```

Fig. 3. (a) Specification (b) Body (c) Instance (d) run-time checks

- CV is a set of clock variables that monitor the time, e.g., CV = {monitorT}.
- IO is a set of input/ouput variables of the program body, e.g., IO = {approach, exit, close, open} ∪ IP ∪ OP ∪ LV ∪ CV in Fig. 1;
- ProgBody is a SARA program body. It is defined as a set of component instances which are interconnected using variables. The connections of component instances are defined as a program configurations, ProgConfigs (definition 2).

Definition 2 (ProgConfig). A program component configuration is defined as a tuple ProgConfig = (CompInsts, CompConnects), where

- CompInsts is a set of component instances (see Definition 3),
- CompConnects is a set of component connections (see Definition 5).

Definition 3 (CompInst). A component Instance is defined as a tuple (InstName, CompTypeName, Priority), where:

- InstName is a user defined name of specific instance of a component type;
- CompTypeName is the name of the corresponding component type, CompType (see definition 4);
- Priority is an integer that defines the execution order of component instances in the context of the component configuration. For instance, the execution order of Fig. 1 is: Gate_Inst_1 < Controller_Inst_1 < Sensor_Inst_1. This means that Sensor_Inst_1 has a highest priority than Controller_Inst_1 and Controller_Inst_1 has a highest priority than Gate_Inst_1.

Definition 4 (CompType). A component type is defined as a tuple CompType = (CompTypeName, IP, OP, LV, Annotation, CompBody), where:

- CompTypeName is the name of the component type, which is defined by its developer. We distinguish two kinds of component type: active and passive components. An active component has its own dedicated thread of execution. While a passive component is directly processed in the context of the calling thread of an active component. Note that a component is either a basic component or a hierarchical component. A hierarchical component contains other components that can be themselves hierarchical or basic (e.g. SARAprog of Fig. 3 (c) that contains basic component instances of Fig. 1). Whereas, a basic component directly encapsulates behavior (e.g., in Fig. 1, BP-component is a basic passive component, and BA-component is a basic active component);
- IP = $\{ip_1, ip_2, ..., ip_n\}$ is a set of input ports;
- OP = $\{op_1, op_2, ..., op_n\}$ is a set of output ports;
- LV is a set of local variables of this component type;
- Annotation is a time annotation of component body operations, e.g., see lines 12-15 of Fig. 3 (a);
- CompBody defines the behavior of instances of component type. The body can be written in any programming language. For example, we use the Ada programming language, e.g., see Fig. 3 (b).

Definition 5 (CompConnect). A component connection is defined as a set of four types of connection, CompConnect = {DC, IC, LC, OC}, where:

- DC is a set of direct connections between IP of component instances and OP of component instances. It is defined as:
  - $DC_n : InstName_i.op_j \rightarrow InstName_k.ip_l$,
  - e.g., $DC_1 : Sensor\_Inst\_1.op_1 \rightarrow Controller\_Inst\_1.ip_1$ in Fig. 1;
- IC is a set of connections that connect input parameters $ip_j \in IP$ of SARAProg to an input port $ip_l \in IP$ of kth component instance or to a program local variable $lv_k \in LV$. It is defined as:
  - IC_n: $ProgName.ip_j \rightarrow InstName_k.ip_l$, or
  - IC_n: $ProgName.ip_j \rightarrow ProgName.vl_k$,
  - e.g., $IC_1 : dist \rightarrow Sensor\_Inst\_1.ip_1$ in Fig. 1 or
  - e.g.,: $IC_2 : init \rightarrow Sensor\_Inst\_1.init$ in Fig. 1;
- LC is a set of connections that involves local variables of a program that do not occur in IC. A local variable $lv_i \in LV$ of SARAProg may be connected to an input port $ip_j \in IP$ of the kth component instance, an output port $op_l \in OP$ of the kth component instance or an output parameters $op_m \in OP$ of SARAProg. It is defined as:
  - LC_n: $ProgName.lv_i \rightarrow InstName_k.ip_j$, or
  - LC_n: $InstName_k.op_l \rightarrow ProgName.lv_i$, or
  - LC_n: $ProgName.lv_i \rightarrow ProgName.op_m$
  - e.g., $LC_1 : reset \rightarrow Controller\_Inst\_1.set$, or
  - e.g., LC_2: $Controller\_Inst\_1.op_3 \rightarrow emergency$ in Fig. 1;
- OC is a set of output connections between $op_i \in OP$ of the kth component instance and output variables $op_j \in OP$ of SARAProg. It is defined as: $OC_n : InstName_k.op_i \rightarrow ProgName.op_j$, e.g.: $OC_1 : Gate\_Inst\_1.op_1 \rightarrow up$ in Fig. 1.

4.2 Time annotations

In this section, we provide predefined time annotations, which are used to annotate our component operations (e.g., lines 12-15 in Fig. 3.(a)). This predefinition facilitates the expression of timing constraints commonly used. While analysing various types of common temporal requirement classifications [7, 12], we found out that most of requirements can be expressed either as a set of obligation rules or as a set of interdiction rules. Table 1 shows some examples of timing response obligation annotations and their descriptions. Generally, in common temporal requirements, an event e, named here *monitored event e* should occur permanently or temporarily in response to a stimulus event, named here *referenced event e'*.

In this paper, we also give to users the possibility to express requirements that refer not only to the timed interval relatively to a given event, but also to the occurrence i^{th} of this event appearance. For example, stating only the timed obligation pattern (e.g., *event e* ***must occur after*** *event e'*) is ambiguous since the assertion does not specify the response time limit within which *e* may occur

Table 1. Time annotations

Time annotations	Descriptions
$between(e, T_{begin}, T_{end}, i, e')$	ensures that a monitored event e must occur within a temporal interval [T_{begin}, T_{end}] after the i^{th} occurrence of referenced event e'.
$mindelay(e, T_{min}, i, e')$	ensures that a monitored event e must occur after a minimum delay T_{min} time unit after the i^{th} occurrence of referenced event e'
$maxdelay(e, T_{max}, i, e')$	ensures that a monitored event e must occur before a maximum delay T_{max} time unit after the i^{th} occurrence of event e'
$exactdelay(e, T, i, e')$	ensures that a monitored event e must occur exactly at a delay T time unit after the i^{th} occurrence of event e'

after e'. In addition, it does not specify if e may occur after the first or the last occurrence of e'. However, affirming that *event e* ***must occur before a maximum delay of 3 time units after the first occurrence*** *of event e'* avoid confusion. In this example, the assertion obtained is identified in Table 1 as the $maxdelay(e, T_{max}, 1, e')$ annotation, where $T_{max} = 3$ time units (e.g., seconds) and $i = 1^{th}$, i.e., the first occurrence of e'.

The goal of these timed annotations is to guide users during the modeling in order to produce a clear and accurate description, while manipulating simple and precise concepts.

4.3 TA model

Timed Automata (TA) is one of the most popular models adapted to real-time system [3]. First, TA models are well adapted for the verification of time properties for real-time components because temporal requirements are explicitly modeled by using state invariants, transition guards and setting or resetting clock variables. Second, a number of methods based on variants of the TA model (or other similar models such as timed Petri nets) have been proposed [11, 13, 14]. In this paper, we use timed automata over input or output actions, called the Timed Automata with Inputs and Outputs (TAIO) [13]. Finally, a number of automatic model checker tools for TA have been efficiently developed, e.g., Uppaal [14] and Kronos [24]. In this work, we use Uppaal as one of these tools for the verification process. It offers a convenient graphical user interface for simulation. In the following the TA system, based on Uppaal TA system, is defined to facilitate the transformation process of the SARA model. This means that all parts which are not used by the transformation process are not included in the definitions.

Definition 6 (TASys). A TA verification system can be defined as a tuple TA = (TAModels, TADeclarations), where:

- TAModels is a set of all TA models used in a system global model. In this work, every TA model is defined according to TAIO (see definition 7);

– TADeclarations is the declaration part that contains all input/output variables of all component instances and all input/output/local variables of the program (see definition 10).

Definition 7 (TAModel). A TA model is defined as a tuple TAModel = (TAName, TASyntax, TASemantic), where:

– TAName is the name of the TA model which appears in the system declarations part to arrange priorities on TAModels;
– TASyntax is the syntax of the state-transition description of TAIO extended with boolean and integer variables (see definition 8);
– TASemantic is the semantic (see definition 9).

Definition 8 (TASyntax). TAIO is represented by the tuple A = (L, l_0, V, Act, Clock, Inv, T), where :

– L is a finite set of locations;
– $l_0 \in L$ is the initial location;
– V = $V_{bool} \cup V_{int} \cup V_{act} \cup V_{const} \cup V_{clock}$ is a finite set of variables (boolean, bounded integer, channel, constant or clock) declared in the TADeclarations part (see definition 10);
– Act = $V_{act} \times \{!, ?\}$ is a set of synchronization actions over channel variables V_{act}. It is the partitioned set of input and output actions, $Act = Act_{in} \cup Act_{out}$. Input actions are denoted a?, b?, etc, and output actions are denoted a!, b!, etc;
– Clock is a finite set of real-valued clocks, $\{x_1, x_2, ..., x_n\}$;
– Inv is a function, that assigns an invariant to each location. $Inv(V_{clock}, V_{int})$ is the set of invariants over clocks $x_j \in Clock$ and integer variables $c \in V_{int}$;
– T is a finite set of edges for transitions.

Each edge T is a tuple (l, g, r, a, l'), where:

– $l, l' \in L$ are respectively the source and destination locations;
– g is a set of time constraints of the form $x \bullet c$, where $x \in Clock$ is a clock variable, $c \in V_{const}$ is an integer constant and $\bullet \in \{<, \leq, =, \geq, >\}$;
– $r \in Clock$ is a set of clocks to reset to zero, ($r := 0$, where 0 is the initial valuation of the clock);
– $a \in Act$ is a set of actions to update ($a := b$, where b is another action).

Definition 9 (TASemantic). The semantic of A = (L, l_0, V, Act, Clock, Inv, T) is defined by the Timed Labeled Transition System (TLTS) [13]. TLTS is a tuple (S, s_0, Act, T_d, T_t), where:

– $S = L \times \mathbb{R}_+^X$ is a set of timed states associated to locations of A;
– $s_0 = (q_0, \vec{0})$ is the initial state. $\vec{0}$ is the valuation assigning 0 to every clock $x \in Clock$ of A;
– T_d is a set of discrete transitions of the form $(s, a, s') = (s', v) \xrightarrow{a} (s', v')$, where $a \in Act$ and there is an edge $E = (l, g, r, a, l',)$, such that v satisfies g and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged; where $t \in \mathbb{R}_+$. T_t must be deterministic.

Definition 10 (TADeclarations). In order to facilitate the transformation process of SARA model to TA model, TADeclarations are partitioned to a TA model declaration part (TAModelDecl) and to a system model declaration part (TASysDecl). TAModelDecl = (dataType, variableName, value), where: - *dataType* is a set of project-specific data types. In this work, we use the Uppaal declaration types: constant, boolean, bounded integer, channels, array or clock; - *variableName* represents the name of the variable and - *value* is considered either the initial value or the constant according to the data type.

TASysDecl defines the execution order by assigning priority to TA models. For example in the example of Fig. 1, input connections (e.g., IC_1) have the highest priority, followed by component instances (e.g., Controller_Inst_1), followed by other connection types according to the execution order defined (e.g., DC_1) and finally followed by output connections (e.g., OC_1).

4.4 Transformation rules

This sub-section presents the transformation rules developed to translate SARA application models to TA models. They are based on the above SARA and TA formal models.

Rule 1 (mapping of declarations). The objective of this rule is to transform the input and output variables of each component instance (CompInst) and all the variables declared in the SARA program (SARAProg) into TA declaration parts (TADeclarations). It is composed of two parts:

- **Rule 1.1** For each CompInst = ($InstName_i$, CompTypeName, Priority), where CompType = (CompTypeName, IP, OP, LV, Annotation, CompBody), insert $ip_j \in IP_i$ and $op_j \in OP_i$ in TADeclarations, where $i = 1, 2, ..., n$ and n is the number of CompType instances, as shown in the left hand of Fig. 4;
- **Rule 1.2** For each $ProgName$ of SARAProg, where $ip_m \in IP$, $op_n \in OP$ and $lv_p \in LV$ insert corresponding variables in TADeclarations, as shown in the right hand of Fig. 4.

```
//Sensor_Inst_1 of Fif. 1 TA declaration
Const int N = 2; // number of trains
typedef int[0,N-1] id; // bounded integer,
clock x1, x2, x3; // sensor clock variables
int ip_1_list_for_dist[N];
bool ip_2_list_for_init[N];
chan op_1_channel_for_approach[N];
chan op_2_channel_for_exit[N];
```

```
//Fig1. program TA declaration
int dist;
bool init;
bool reset;
...
...
bool up;
bool down;
```

Fig. 4. Example of Fig.1 TAdeclarations

Rule 2 (mapping of CompInsts). The objective of this rule is to transform CompInsts to TAModels. For each CompInst = (InstName, CompTypeName, Priority), where CompType = (CompTypeName, IP, OP, LV, Annotation, CompBody), insert the corresponding TAModel to TASys with TAName

= InstName by using a user predefined TAModel library and by taking into account the annotations in order to add the suitable state invariants and transition guards according to rule 4. For example, see Fig.7, which corresponds to our use case TAModel corresponding to the three component instances of Fig. 1.

Rule 3 (Mapping of annotations). This rule is invoked from rule 3 when a component type contains time annotations as shown in Table 1, and illustrated in lines 12-15 of Fig. 3 (a). These annotations are translated into boolean conditions in TASyst by respecting the TASemantic shown in Definition 8.

Rule 4 (mapping of connections). The objective of this rule is to transform connections. For each connection in CompConnect = (DC, IC, LC, OC), a TAModel in TASys is inserted by respecting the following rule parts:

- **Rule 4.1** For each $DC_n : InstName_i.op_j \rightarrow InstName_k.ip_l$, insert a TAModel $(L, l_0, V, Act, Clock, Inv, T)$ with name DC_n, where $L = \{l_0\} = \{DC_n\}$, $V = \{op_j, ip_l\}$, $Inv = \{\}$, and $T = \{(q, g, r, a, q')\}$, with $g = \{op_j \oplus ip_l\}$, where $\oplus$ or XOR represents the inequality function between the output op_j and the input ip_l, $r \in Clock \wedge r = \{\}$ and $a \in Act \wedge a = \{ip_l := op_j\}$ (e.g., DC_3 in Fig. 5);

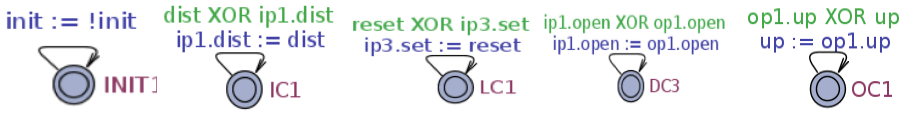

Fig. 5. Example of Fig.1 TA connections

- **Rule 4.2** For each $IC_n : ProgName.ip_j \rightarrow InstName_k.ip_l$ or $ProgName.ip_j \rightarrow ProgName.vl_k$, insert a TAModel with name IC_n (e.g., IC_1 in Fig. 5);
- **Rule 4.3** For each $LC_n : ProgName.vl_i \rightarrow InstName_k.ip_j, InstName_k.op_l \rightarrow ProgName.lv_i$ or $ProgName.lv_i \rightarrow ProgName.op_m$, insert a TAModel, with name LC_n (e.g., LC_1 in Fig. 5);
- **Rule 4.4** For each $OC_n : InstName_k.op_i \rightarrow Program.op_j$, insert a TAModel with name OC_n, (e.g., OC_1 in Fig. 5);

Rule 5 (Initial input mapping). The objective of this rule is to allow manual validation by using TASys like Uppaal simulator. For this, input variables IP of the SARA program are allowed to be changed by the user. For each $ProgName.ip_j$ in IP, insert TAModel with name $INIT_n$, where $L = \{INIT_n\}$, $V = \{ip_j\}$ and $T = \{(q, g, r, a, q')\}$, with $a \in Act \wedge a = \{ip_j := !ip_j\}$, $r \in Clock \wedge r = \{\} \wedge g = \{\}$, (e.g., $INIT1$ in Fig. 5).

Rule 6 (mapping of execution flow). The objective of this rule is to define the execution flow of the TASys by using priorities on TA models. Based on priority defined in CompInst = (InstName, CompTypeName, Priority) shown in Definition 3, assign priority to each TAName in TASysDecl (see definition 10).

The validation of these transformation rules are realized through their application in some use case scenarios of SARA model. The preservation of syntax and semantic information across the transformations was checked whenever the TA output models are successful processed by the Uppaal simulation tool.

5 Proof of concept

In this section, a simulation scenario of a use case is translated into the TA model for the verification of system-level requirements presented in Section 2.

5.1 Use case

The safety of rail-road Level Crossing (LC) has long been a major concern for railway and road stakeholders since LC accidents often generate serious material damage, traffic disruption and human losses. As a consequence, the LC system has already been used as a benchmark in several previous verification approaches [4, 15]. Fig. 6 shows the LC topography considered in this paper. It is composed of the following features: (1) double-track railway lines (*UpLine* and *DownLine*); (2) roads with traffic in both directions; (3) traffic lights to manage the road traffic in the LC zone; (4) sound alarms to signal train arrival; (5) two half-barriers used to prevent road users from crossing while trains are passing; (6) three train sensors An_i, Ap_i and Ex_i in both track lines. For example, in *DownLine*, the An_2 is the anticipation sensor, which allows the detection of the approaching train speed, necessary to alert road users with sound alarm and road lights. The sensor Ap_2 is used to detect the arrival of trains in the LC zone and the exit sensor Ex_2 is used to announce the departure of trains after exiting the LC zone. Since several trains with different speeds (passenger or freight trains) can circulate on railway lines, the required durations between sensors are expressed with intervals in Fig. 6. For example, $d_1 = [10, 15]$ second (s) is a required interval of durations between An_i and Ap_i. This interval and the others must be respected by trains circulating in the railway track lines.

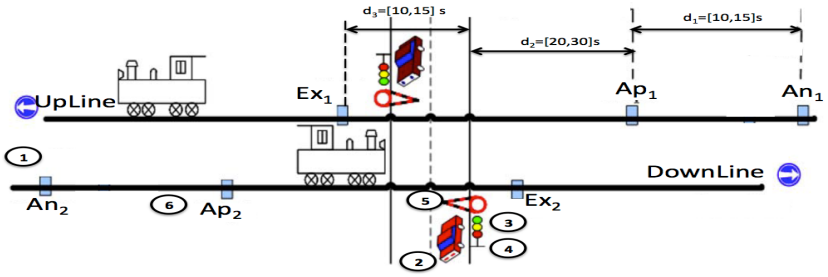

Fig. 6. Level crossing topography

5.2 Transformation of simulation scenario to TA model

In this step, the SARA model of a simulation scenario is translated into the TA model. Fig. 1 and 3 present an architecture and an implementation of a simulation example of our use case by using software components (Sensor, Controller, Gate). These components are executed in parallel and are synchronized through various events, e.g., *appr*, *close*, etc, in order to provide the automatic LC control system. This LC model is manually transformed to the Uppaal TA model

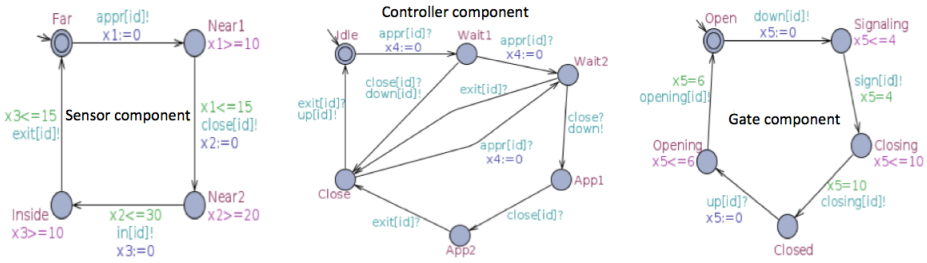

Fig. 7. The level crossing TA model: Controller model, Sensor model and Gate model

in oder to use its simulation tool for the verification of our requirements. Figure 7 shows the Uppaal TA model of a LC system scenario, presented as follows.

When a train arrives in the monitored area, it activates the first sensor instance An_i, (i.e., An_1 for the *UpLine* direction and An_2 for the *DownLine* direction) and it sends the *appr* event with its *id* (*appr*[*id*]! in Fig. 6) to the controller. In the same way, when it approaches the crossing section, it activates the second sensor Ap_i and sends the *close*[*id*]! event to the controller. The train spends at least 10 *s* and at most 15 *s* in this first section (between An_i and Ap_i, i.e., between *appr* sending and *close* sending). This timing requirement is presented as an invariant of state $Near1$ (i.e., $x1 >= 10$ in Fig. 6) and the guard of a transition to state $Near1$ to $Near2$ (i.e., $x1 <= 15$). The train leaves the crossing section at least 30 *s* and at most 45 *s* after sending the *close* event. When a train leaves the crossing section, it activates the third sensor instance Ex_i, and the train sends an *exit*! signal to the controller. When the *close*? signal (respectively *exit*?) is received, the controller immediately sends a *down*! signal (respectively *up*!) to the gate. We assume that there is no overlap between trains in the same direction, which means that the controller handles at most two trains at the same time, i.e., at most one train in each direction. The controller model of Fig. 7 is a simplified version of the controller behavioral model. It deals with the case when the gate is closed and when there is a train approaching meanwhile, the controller decides to open immediately the gate or to wait certain duration before open the gate. The Gate responds to *down*? signal by moving down and takes 10 *s* to be completely closed. Indeed, it takes 4 *s* to activate the light warning the vehicles approaching the LC and 6 *s* to close the gate. Conversely, it responds to *up* signal by moving up and it takes 6 *s* to be completely open.

5.3 Requirement validation using observer patterns

The verification consists in checking that the parallel composition of the application model under test and its safety requirements observers never enters an erroneous state. The system-level safety requirements stated in Section 2 are checked based on the predefined observer patterns.

Requirement 1 validation. This requirement is a critical condition aiming to avoid train-car collisions in the crossing zone. It states that "The gate is *never open* when the train is *inside*". This requirement will be expressed as an *exclusion pattern* between *open* state and *inside* state. Firstly, this textual description of the requirement, which is presented as an annotation in the SARA component

model (see Fig. 3 (a)), is intuitively presented as an exclusion observer pattern in the TA model. Fig.8 (a) presents the graphical representation of this exclusion pattern, which is used to check that a given situation (state $S1$ and $S2$ are activated at the same time with event $b1?$ and $a1?$) is never reached.

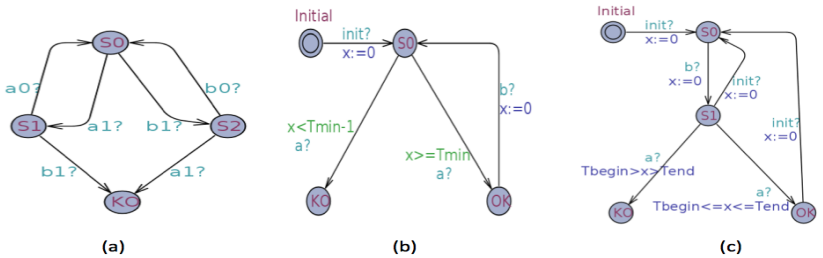

Fig. 8. (a) Exclusion observer pattern, (b) Forbidden before observer pattern, (c) Obligation between observer pattern

Secondly, once identified, the patterns are instantiated with the appropriate parameters. In our case, the exclusion pattern is instantiated with $in[id]$, $exit[id]$, $opening[id]$ and $down[id]$ events, instead of $a1$, $a0$, $b1$ and $b0$, respectively. Thirdly, once generated, the TA patterns are synchronized with the system model to generate a global system. Finally, the verification of our use case is carried out on the Uppaal model checker. The "KO" node in the TA observers is never reached, which means that this requirement "the gate must be down when the train is inside the crossing" is always evaluated to true.

Requirement 2 validation. In our scenario model shown in section 5.2, we firstly supposed that the Gate should stay in the *open* state at least 15 s before becoming closed again. According to the Gate model in Fig. 7, the Gate takes 4 s to signal when vehicles can traverse, and 6 s to be closed. So, the time that separates the *up* and *down* detection is $Tmin$ = 15 s - 4 s. This means that the *down* detection should be done at least 11 s after the *up* detection. With the Uppaal model checker, the "KO" node of Fig. 3 (b) observer pattern instantiation ($down[id]$ instead of a and $up[id]$ instead of b) is reached. Indeed, the path that violates the requirement can be expressed as follows: (1) the first train leaves the critical section and sends $exit[id1]$ signal. Exit detection triggers the $up[id1]$ sending. Consequently, the gate is open; (2) suppose that a second train is simultaneously entering the LC section, an $appr[id2]$ signal is detected; (3) the second train takes 10 s to trigger the close signal, which triggers the $down[id2]$ sending; (4) As a result, 10 s is computed between $up[id1]$ and $down[id2]$, which violates this requirement :"when the gate is opened to road traffic, it must stay open at least T_{min} time units, where $T_{min} = 11$ s". This verification result helps the designer to search the accepted time parameter between open cycles.

Requirement 3 validation: "once closed and when there is no train approaching meanwhile, the gate must be kept closed at least T_{begin} and at most T_{end}." For the validation of this requirement, we determine the different speed intervals allowed in the track lines. In the beginning, the train speed interval considered in our double-track railway lines of Fig. 6 is [14,45] m/s. The counter examples given by Uppaal model checker, allow to identify new speed intervals

that validate the requirement, and so on. The obtained speed intervals that validate this requirement an above requirements are: {[14, 15[, [15, 16[, [16, 18[, [18, 20[, [20, 22[, [22, 30[, [30, 45]}.

5.4 Proof of concept discussion

The first results obtained in the previous "proof in use" are encouraging and show the correctness of the defined rules. However, the more formal validation still need to be defined in order to formally verify that the syntax and specially the semantic information are indeed preserved across the transformation.

Having said that, the strong goal of our approach is to express and verify requirements relative to certain scenarios of use cases. The scenario-based description, rather the entire system description, allows a limitation of the explored space search, and hence a first reduction in the combinatorial explosion, which is an important limitation for the application of model checking techniques in complex software projects [23]. For this reason, the strong assumptions we made about the SARA model is that the designer is able to identify all possible interactions between components of the system and between the system and its environment. We justify this strong hypothesis, particularly in the field of embedded systems, by the fact that the designer of a component needs to know precisely and completely the context, i.e., constraints, conditions, of its system for properly developing it. It would be necessary to study formally the validity of this working assumption for scalability in the targeted applications. In this paper, we do not address this aspect, which is planned for our future work.

6 Related work

Automating the verification process of applications increases development productivity and quality [1]. There are several research works in this direction. These works are mainly based on the transformation of the source models to the target formal models, which are next used for verification purposes by exploiting verification tools [18]. For example, Solimaan et al. transform function block diagram to timed automata for the automated formal verification by using the Uppaal model checker [19]. Textual safety requirements are converted to CTL properties and are checked on the Uppaal TA system using the verifier tool. This verification process requires significant knowledge of higher order logic and theorem proving. This process has two main limitations. The first one is that users must be familiar with the higher order logic in CTL. The second is the lack of patterns for high-level system properties. In contrast, in our verification methodology, we use observer-based verification by providing the timed annotation patterns which promote reusability. As demonstrated in [4], the verification of safety properties by using observer-based verification does not require learning another language for the purpose of property specification. The verification task can be reduced to a simple reachability analysis. Our method suggests using generic predefined observation patterns [6] to check the temporal requirements of a given system.

In this work, we focus on the verification approach that takes advantage of the flexibility of the source model and the analysis facilities offered by a target formal model. In the same way, Mekki et al., based on the flexibility an the expressiveness of UML State Machine (UML SM), transform this semi-formal model to the TA model [15]. This method allows the automated verification of temporal requirements, initially expressed in a semi-formal formalism, through the model transformation. This work is focused on the validation of new functional requirements that prevent several accidents at LCs with model-checking techniques [15]. We use the suggested new LC topography as our use case. In contrast, our work focuses on the integrated development approach. Indeed, given a software requirement specification of safety-critical software, the proposed development process is to guide developers at the first design stage for the identification of requirement types, for the modeling of requirements and for the verification of requirement models before implementation.

7 Conclusion

The main challenge we face in this paper is how to transform a source model of safety-critical applications to a target model suitable for automatic formal verification. In order to face this challenge, we formalize our SARA component model and the TA model. Based on these formal models, transformation rules were then defined. A component model of a simulation scenario is manually transformed to the Uppaal TA model to validate some safety requirements. The counter examples discovered during the verification process can help the developer to identify the software components that should be modified before the implementation and the integration. After the verification phase, the scenario model is implemented with the Ravenscar profile of Ada language, which is one of the recommended languages in the development of railway safety-critical applications. The complete process to validate the safety requirements shows the understanding of the transformation process and the applicability of the proposed approach.

This is very encouraging to automate the transformation of our SARA model to the TA model. We are currently working on the development of this automation. As a consequence, our future work targets the automation process for the automatic verification of timing requirement annotations, which are not supported by the annotation checking tool [9], used in this paper. In addition, the application of our approach to several use case scenarios is another direction to demonstrate the efficiency and the scalability of our approach.

Acknowledgements This work is supported by IFSTTAR Institute and ANR VEGAS Project.

References

1. R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(2):20:1–20:39, January 2011.

2. M. Akerholm, A. Moller, H. Hansson, and M. Nolin. Towards a dependable component technology for embedded system applications. In *10th International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 320–328, Feb 2005.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.
4. Z. Bhatti, R. Sinha, and P. Roop. Observer based verification of iec 61499 function blocks. In *Industrial Informatics (INDIN)*, pages 609–614, July 2011.
5. I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5):593–615, 2011.
6. J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi. Timed automata patterns. *Software Engineering, IEEE Transactions on*, 34(6):844–859, Nov 2008.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of ICSE'99*, pages 411–420.
8. EN-50128. *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, January 2011.
9. Hi-lite. Simplifying the use of formal methods: verification by contract. `http://www.open-do.org/projects/hi-lite/`.
10. IEC-61499. *IEC 61499 function blocks for industrial-process measurement and control systems*. Geneva, Switzerland, 2005.
11. D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.
12. S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th ICSE*, pages 372–381, 2005.
13. M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.
14. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
15. A. Mekki, M. Ghazel, and A. Toguyeni. Validation of a new functional design of automatic protection systems at level crossings with model-checking techniques. *IEEE Transactions on Intelligent Transportation Systems*, 13(2):714–723, 2012.
16. M. Sango. Application of sara approach to ertms/etcs on-board train control system. Technical Report, IFSTTAR, April 2013. `http://urls.fr/sara`.
17. M. Sango, C. Gransart, and L. Duchien. Safety component-based approach and its application to ERTMS/ETCS on-board train control system. In *TRA2014 Transport Research Arena 2014*, Paris, France, April 2014.
18. S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003.
19. D. Soliman, K. Thramboulidis, and G. Frey. Transformation of function block diagrams to uppaal timed automata for the verification of safety applications. *Annual Reviews in Control*, 36(2):338 – 345, 2012.
20. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
21. G. Tamura, R. Casallas, A. Cleve, and L. Duchien. Qos contract-aware reconfiguration of component architectures using e-graphs. In *FACS'10*, pages 34–52.
22. K. Taylor. Addressing road user behavioural changes at railway levelcrossings. In *ACRS-Travelsafe National Conference*, pages 368–375, Brisbane, Australia, 2008.
23. J. Whittle. Specifying precise use cases with use case charts. In *MoDELS Satellite Events*, pages 290–301, 2005.
24. S. Yovine. A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.